Measuring argument passing in Go and C++

Raphael kena Poss

August 2018

Contents

> **Note**
> The latest version of this document can be found online at https://science.raphael.poss.name/measuring-argument-passing-in-go-and-cpp.html. Alternate formats: Source, PDF.

Introduction

The following document investigates the performance of passing arguments to functions in Go, and compare that to C++.

This is a follow-up to an earlier article I wrote on the Go calling convention on x86-64. It provides some quantitative insight into the observations made in that earlier analysis.

We focus here on just one of the aspects of the Go calling sequence: passing arguments to functions and how the number of arguments influences performance.

Further analysis of the other aspects of calling sequences (return values, errors, exceptions) are out of scope and I will revisit these topics later.

> **Note**
> This document was designed as Jupyter Notebook. The notebook and accompanying data files can be downloaded here.

Experimental setup

The source code for the experiments can be found here: http://github.com/knz/callbench. There are multiple experiment sources in that repository, today we are focusing on the `nargs` experiments: comparing the result of passing varying number of arguments to Go and C++ functions.

The functions we are going to measure look like this:

- in Go,

```
//go:noinline
func f4(a0,a1,a2,a3 int) int { return a0+a1+a2+a3 }

func BenchmarkArg4(b *testing.B) {
 val := 1
 for i := 0; i < b.N; i++ {
  val += f4(i,i,i,i);
 }
 CONSUME(b, val);
}
```

- in C++,

```
__attribute__((noinline))
long f4(long a0,long a1,long a2,long a3) { return a0+a1+a2+a3; }
void BenchmarkArg4(B* b) {
  long val = 1, N = b->N;
  for (long i = 0; i < N; i++) {
     NOP(i);
     val += f4(i,i,i,i);
  }
```

```
  CONSUME(b, val);
}
```

There are 21 variants of these benchmarks, from 0 to 20 arguments passed. They are generated by the accompanying `gen_nargs.sh` scripts.

In every case, the function sums its arguments and returns the result. I chose the addition as this is probably the cheapest operation that can be performed on the processor and so will not dominate the cost of function calls. Both versions use the native 64-bit integer type, called `int` in Go and `long` in C++.

The functions are marked as `noinline` to ensure they are not inlined in the benchmark kernel, and the call sequence in machine code eliminated, by optimizations.

The benchmark kernel uses in Go the standard `go test -bench` infrastructure. In C++, it uses the library cppbench which re-implements the `go test -bench` infrastructure in C++, so that we are measuring the same things in the same way in both languages.

You will probably notice that the benchmarking kernel must use a special `CONSUME()` function/macro. We must do this in both languages to ensure the result of the computaiton by the benchmarking loop is eventually used. Otherwise, the compiler will see the value is dead and eliminate the computation altogether.

In addition, in C++ we also must use a special `NOP()` mqcro that invalidates the compiler's knowledge about the value of the loop counter; otherwise, the compiler would peek into the callee function `fN` and pre-compute the result of the entire loop using just one multiplication (the Go compiler knows no such optimizationyet).

Finally, in addition to both series of 21 variants using named arguments, we also define 21 more variants that pass the arguments to functions that accept a variable argument list:

- in Go,

```
//go:noinline
func fvar(args ...int) int {
      val := 0
      for _, a := range args {
              val += a
      }
      return val
}
```

- in C++,

```
__attribute__((noinline))
long fvar(size_t nargs, ...) {
      va_list ap;
      va_start(ap, nargs);
      long val = 0;
      for (size_t i = 0; i < nargs; i++) {
              val += va_arg(ap, long);
      }
      va_end(ap);
      return val;
}
```

Note

This C++ code uses traditional C-style variable argument lists, where a single compiled function can accept different numbers of arguments from different callers.
This is different from new-style variadic template functions where each call instantiates a variant of the function with a different number of *fixed* arguments.
Variadic template functions are compiled like functions with a fixed number of arguments and thus have the same performance profile, and I will not consider them further here.

We run all these 84 benchmarks as follows:

- `cd go; make nargs_go.log`
- `cd cpp; make nargs_cpp.log`

Then we copy the two log files into the directory of the Jupyter notebook.
The specific measurements illustrated in the rest of this document were obtained in the following environment:

- CPU: AMD Ryzen 7 1800X 3593.33MHz Family=0x17 Model=0x1 Stepping=1
- OS: FreeBSD 12.0-CURRENT r336756
- go version 1.10.3 freebsd/amd64
- C++ FreeBSD clang version 6.0.1 (tags/RELEASE_601/final 335540)

Data preparation

Data transformation

The benchmark results are mingled with other test outputs. Let's extract every line starting with `Benchmark` and then remove the `Benchmark` prefix:

```
def load(fname):
    data = [x for x in open(fname).read().split('\n') if x.startswith('Benchmark')]
    data = [x[9:] for x in data]
    return data

data = ['Go/' + x for x in load('nargs_go.log')] + \
       ['Cpp/' + x for x in load('nargs_cpp.log')]
print ("number of results:", len(data))
print ("example result row: %r" % data[0])

number of results: 84
example result row: 'Go/Arg0           2000000000               0.28 ns/op'
```

Each row is composed of the benchmark name, some counter of how many times the benchmark framework had to run the benchmark loop to stabilize the measurement, and the measurement itself.

We'll want just the name and the numeric value of the measurement - a number of nanoseconds:

```
import re
r = re.compile('^(\S+)\s+\S+\s+(\S+)\s+.*')
data = [m.groups() for m in [r.match(x) for x in data] if m is not None]
print ("example result row: %r" % (data[0],))

example result row: ('Go/Arg0', '0.28')
```

Notice how the value part is still a Python string. This won't do! Let's make it a floating-point number.

```
data = [(x[0], float(x[1])) for x in data]
print("example result row: %r" % (data[0],))

example result row: ('Go/Arg0', 0.28)
```

Data filtering / extraction

The numbers in each benchmark name will make good values for x-axes. But they are still strings at this point. We'll need a function that can get a x/y data series from a filter on the benchmark name.

```
def filterdata(pattern):
    r = re.compile(pattern)
    matchvals = [(r.match(x[0]), x[1]) for x in data]
    xvals = [int(i[0].group(1)) for i in matchvals if i[0] is not None]
    yvals = [i[1] for i in matchvals if i[0] is not None]
    return xvals, yvals
```

And use it to filter our data:

```
xgo, ygo = filterdata('Go/Arg(\d+)')
print("data series:")
print(xgo)
print(ygo)

# Really the X values are the same for all series.
xvals = xgo
_, ycpp = filterdata('Cpp/Arg(\d+)')
_, ygovar = filterdata('Go/VarArg(\d+)')
_, ycppvar = filterdata('Cpp/VarArg(\d+)')
print (ycpp, ygovar, ycppvar)

data series:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
[0.28, 2.53, 2.66, 2.88, 3.02, 3.14, 3.16, 3.34, 3.64, 3.89, 4.17, 4.45, 4.73, 5.06, 5.3, 5.6, 5.79, 6.06,
[1.65, 1.37, 1.38, 1.38, 1.38, 1.65, 1.38, 1.65, 1.65, 1.93, 1.71, 2.01, 2.07, 2.41, 2.5, 2.82, 3.03, 3.34,
```

Yay!

Data analysis

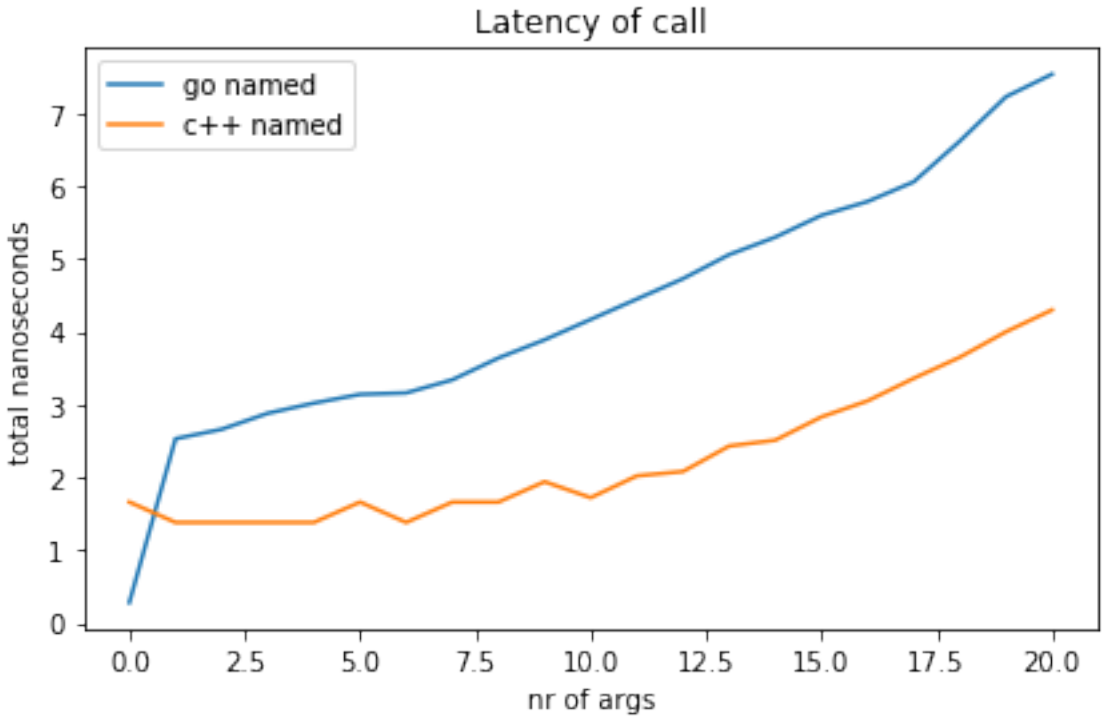
We'll use `matplotlib` for plotting. This needs to be initialized first:

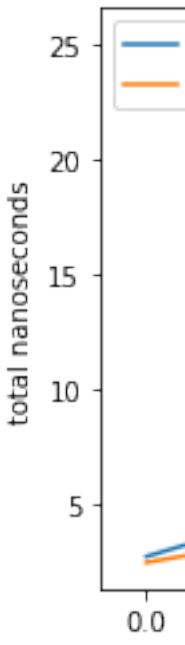
```
%matplotlib inline

import matplotlib
import matplotlib.pyplot as plt
```

We can compare our benchmark results for the two `Arg` and `VarArg` functions by plotting their x- and y-series extracted above:

```
plt.figure(figsize=(15, 4))
plt.subplot(1,2,1)
plt.plot(xvals, ygo, label='go named')
plt.plot(xvals, ycpp, label='c++ named')
plt.title('Latency of call')
plt.xlabel('nr of args'); plt.ylabel('total nanoseconds')
plt.legend()
plt.subplot(1,2,2)
plt.plot(xvals, ygovar, label='go varags')
plt.plot(xvals, ycppvar, label='c++ varargs')
plt.title('Latency of call')
plt.xlabel('nr of args'); plt.ylabel('total nanoseconds')
plt.legend()
plt.show()
```





What does this tell us?

- the benchmark that passes zero named arguments in Go has something funky. Indeed, if we peek at the generated machine code, we see that the Go compiler has bypassed the function call: it was able to peek that the function always returns zero and avoid calling it. This is why the data point is so low.

- the Go calling convention for **named arguments** is about twice as slow as the C++ one. This is expected since Go passes arguments using memory and C++ using registers..
- passing 10 arguments through a variable argument list costs about the same in C++ as in Go. However, in C++ it is about 5 times slower than 10 named arguments. In Go, it is about 2 times slower.
- the C++ calling convention for **variable argument lists** is faster than Go's until about 6 arguments, then slower afterwards. The fact the first 6 are faster is because they are passed into registers (where Go would use memory instead). However beyond 6 arguments, C++ uses memory too and the call sequence is more complex than for Go.

Let's dive deeper. What is the marginal cost of adding more arguments?

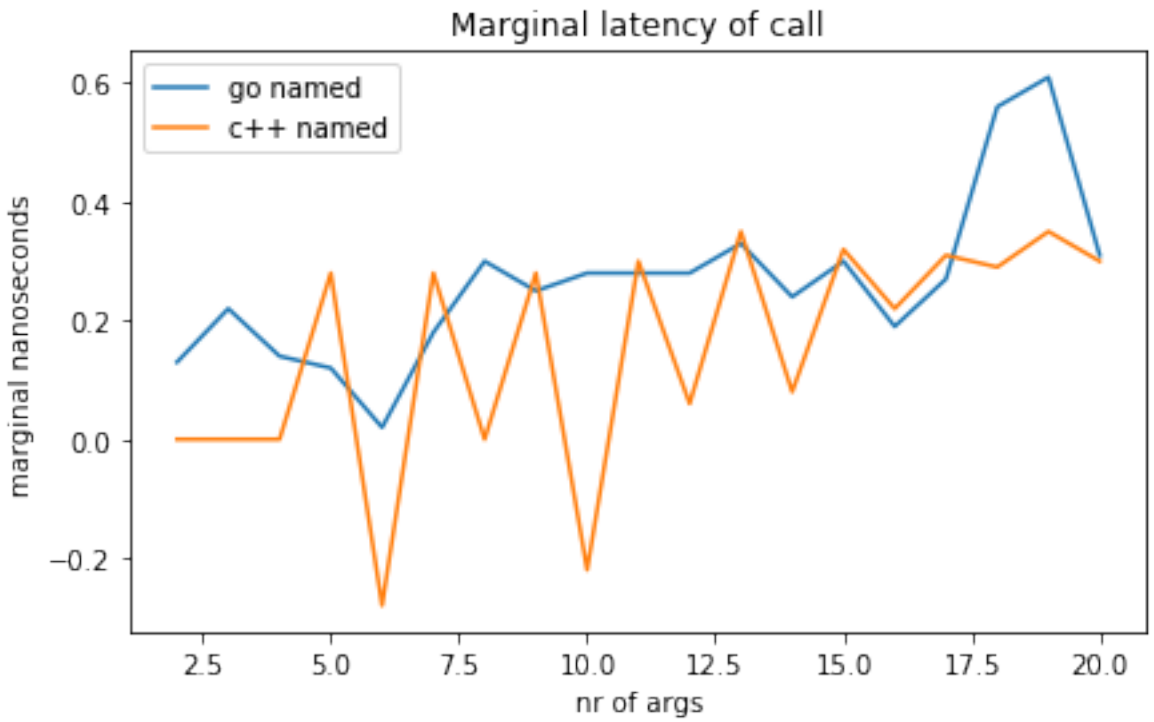
We'll look at the data from 1 argument onwards, because the data point for zero arguments in Go is invalid.

```
xvalsm = xvals[2:]

def marginals(yvals):
    return [y - yvals[i] for (i, y) in enumerate(yvals[1:])]

ygom = marginals(ygo[1:])
ygovarm = marginals(ygovar[1:])
ycppm = marginals(ycpp[1:])
ycppvarm = marginals(ycppvar[1:])

plt.figure(figsize=(15, 4))
plt.subplot(1,2,1)
plt.plot(xvalsm, ygom, label='go named')
plt.plot(xvalsm, ycppm, label='c++ named')
plt.title('Marginal latency of call')
plt.xlabel('nr of args'); plt.ylabel(',arginal nanoseconds')
plt.legend()
plt.subplot(1,2,2)
plt.plot(xvalsm, ygovarm, label='go varags')
plt.plot(xvalsm, ycppvarm, label='c++ varargs')
plt.title('Marginal latency of call')
plt.xlabel('nr of args'); plt.ylabel(',arginal nanoseconds')
plt.legend()
plt.show()
```



The marginal cost of adding arguments in Go is always positive, between 0 and 0.4 nanoseconds per argument for named arguments, and somewhere between zero and 1 nanosecond per argument for variable argument lists. This is expected as each argument adds two memory operations and memory operations are always computationally more expensive.

There is something strange happening in C++ though. There is no marginal cost to go from 2 to 3 arguments. This is true even though the generated machine code does contain 1 more instruction in the caller (to prepare the argument) and 1 more in the callee (to use it).

Then when going from 2 to 3 there is an increase, but it is cancelled out going from 3 to 4, so that passing 4 arguments is really as expensive as just 1. Even 6 arguments is as expensive as just 1, according to the benchmark. And all the while the generated machine code is 6 times longer with 6 arguments than just 1!

What is going on? To understand this, we should recall that the benchmark function executes the calls in a loop. The measurement per call obtained by the framework is really the amortized mean latency, i.e. total measurement time over many calls, divided by the number of calls.

The salient point here is that there are many calls to the same function, with the same control path. This enables micro-architecture optimizations of this particular CPU to kick in: **branch prediction and superscalar scheduling** (multiple instructions enter the pipeline side by side) make this CPU able to process the instructions of one call in parallel with the instructions of the next call, so they overlap in the CPU pipeline. As a result, the benchmark time computation mistakenly determines each call's latency to be shorter than it really is.

The reason why this hardware optimization is available to the C++ code and not (or not as much) to the Go code is that superscalar instruction issue is usually optimized for register-register computations, and prevented by dependent memory accesses (i.e. when subsequent instructions use the same memory location). Even when the micro-architecture can overlap memory accesses, the max number of in-flight memory instructions is typically smaller than that of register-register

instructions. **The choice Go makes for memory-based argument passing thus prevents Go program from benefiting fully from modern micro-arch optimizations in hardware.**

To get accurate measurements of the exact non-pipelined latency would require peeking into the CPU's cycle counter in between iterations of the benchmarking loop. This is readily possible in C++ (with inline assembly and the `rdtsc` instruction) but I have not yet found a way to do this in Go. Because I want to keep both sets of benchmarks equivalent, I will thus do neither here.

Summary and conclusions

On an off-the-shelf, relatively modern x86-64 implementation by AMD, the calling convention of Go makes Go function calls perform twice slower than the equivalent C++ code.

This performance difference is caused mainly by the choice of Go to use memory to pass arguments (a choice unlikely to be revisited any time soon, as per the discussion on a proposal to change it).

Using memory causes Go to use more machine instructions for equivalent calls, which incurs a penalty of its own in any case. Unfortunately, using memory also prevents certain micro-architectural optimizations to kick in in the CPU hardware (in particular superscalar issue). These optimizations give an extra boost to the machine code compiled from C++ and other languages that properly utilize register-based argument passing.

To put these observations into context, one should keep in mind that the cost of function calls is usually amortized by sufficient code inside the function. If the Go compiler is able to optimize the *body* of a function just as well as a C++ compiler (a pretty big assumption, but let's take it for the sake of this argument), a sufficiently large function body can offset the cost of the calling sequence and erase that particular difference between the two languages.

Furthermore, for small functions even though the body would then be insufficiently large to offset the calling sequence, one should remember that the benchmarks considered here *disable inlining* by the compiler. In real-world code, inlining is enabled and small functions are often picked for inlining. Once the code is inlined in the caller, the call sequence is eliminated and the difference between the two languages can be erased again.

The situation is perhaps more interesting when the function body is not quite large enough to offset the cost of the calling sequence, and the function cannot be inlined. This commonly happens with recursive functions or dynamic dispatch (virtual calls in C++ / interface calls in Go). We will revisit these topics in a later analysis.

Also in the series:

- The Go low-level calling convention on x86-64
- Measuring multiple return values in Go and C++
- Measuring errors vs. exceptions in Go and C++

Copyright and licensing

SC fingerprint: `fp:SWkXsUsBWw049Vthc58TLJFPaq_TSNwagRmBRjD4yTMEqA`